Homework 5

Due: Wednesday, May 7th, 11pm

While working on this homework, it may be useful to refer to the first page of the Section 5 worksheet where we have a list of function definitions for easy access. As usual, the section tasks are also good practice for the tasks in this homework.

Submission

After completing all parts below, submit your solutions on Gradescope. The collection of all written answers to problems described in this worksheet should be submitted as a PDF to **"HW5 Written"**.

Don't forget to check that the submitted file is up-to-date with all written work you completed!

If you're using LaTeX, please make sure your file compiles correctly. You may handwrite your work (on a tablet or paper) or type it, provided it is legible and dark enough to read. When you turn in on Gradescope, please match each HW problem to the page with your work on it. If you fail to have readable work or assign pages, you will receive a point deduction.

List Notations

There are two ways you may have seen the type for lists defined. One way is List and the other way is List$\langle T \rangle$. In this homework and onward, we'll be using List$\langle T \rangle$ where T should be replaced with the type of the elements in the List. For example, List$\langle \mathbb{Z} \rangle$ is for a list of integers, List$\langle \mathbb{N} \rangle$ is for a list of natural numbers, etc.

Written

Task 1 – Twice to Meet You **[16 pts]**

The functions twice-evens and twice-odds, both of type $\text{List}\langle\mathbb{Z}\rangle \to \text{List}\langle\mathbb{Z}\rangle$, take a list as input and return the list where the numbers at even and odd indexes, respectively, are doubled, and the others are left as is. We can define these recursively, in terms of each other ("mutual recursion"), as follows:

$$\begin{aligned} \text{twice-evens}(\text{nil}) &:= \text{nil} \\ \text{twice-evens}(x :: L) &:= 2x :: \text{twice-odds}(L) \end{aligned}$$

$$\begin{aligned} \text{twice-odds}(\text{nil}) &:= \text{nil} \\ \text{twice-odds}(x :: L) &:= x :: \text{twice-evens}(L) \end{aligned}$$

The function twice of type $\text{List}\langle\mathbb{Z}\rangle \to \text{List}\langle\mathbb{Z}\rangle$ which doubles every element in the list:

$$\begin{aligned} \text{twice}(\text{nil}) &:= \text{nil} \\ \text{twice}(x :: L) &:= 2x :: \text{twice}(L) \end{aligned}$$

The function sum of type $\text{List}\langle\mathbb{Z}\rangle \to \mathbb{Z}$, take a list as input and returns the sum of all elements:

$$\begin{aligned} \text{sum}(\text{nil}) &:= 0 \\ \text{sum}(x :: L) &:= x + \text{sum}(L) \end{aligned}$$

Now, suppose that you see the following snippet of TypeScript code. The code in the snippet uses `len`, `sum`, `twice_evens`, and (implicitly) twice_odds, all of which are TypeScript implementations of the mathematical functions with the same names.

```
const x = sum(twice_evens(L));
const y = sum(twice_odds(L));

if (len(L) === 2n)
  return x + y;  // = sum(L) + sum(twice(L))
```

The comment shows the definition of what should be returned, but the code is not a direct translation of that. Below we will use reasoning to prove that the code is correct.

Note that, if $\text{len}(L) = 2$, then $L = a :: b :: \text{nil}$ for some integers a and b, as we have previously proven in section. We will use that below.

(a) Using the fact that $L = a :: b :: \text{nil}$, prove by calculation that $\text{sum}(L) = a + b$.

(b) Using the same fact, prove by calculation that $\text{sum}(\text{twice}(L)) = 2a + 2b$

(c) Using the same fact, prove by calculation that $\text{sum}(\text{twice-evens}(L)) = 2a + b$.

(d) Using the same fact, prove by calculation that $\text{sum}(\text{twice-odds}(L)) = a + 2b$.

(e) Prove that the code is correct by showing that `x + y = sum(L) + sum(twice(L))`, i.e., that

$$\text{sum}(\text{twice-evens}(L)) + \text{sum}(\text{twice-odds}(L)) = \text{sum}(L) + \text{sum}(\text{twice}(L))$$

You are free to cite parts (a-d) in your calculation since we know that $L = \text{cons}(a, \text{cons}(b, \text{nil}))$ holds on the line with the return statement. (You can write, e.g., "part (a)" as your explanation on the line that uses the fact proven in part (a).)

Task 2 – Good Swap, Bad Swap **[14 pts]**

This problem uses the following function, swap : $\text{List}\langle\mathbb{Z}\rangle \to \text{List}\langle\mathbb{Z}\rangle$, that swaps adjacent values in a list:

$$\begin{aligned}\text{swap}(\text{nil}) &:= \text{nil}\\ \text{swap}(a :: \text{nil}) &:= a :: \text{nil}\\ \text{swap}(a :: b :: L) &:= b :: a :: \text{swap}(L)\end{aligned}$$

Lists of length 0 and 1 are left as is, and for lists of length 2 or more, the order of the first two elements are swapped before we recurse on the remainder of the list following those two elements.

Suppose you see the following snippet in some TypeScript code. It uses `len` and swap, which are TypeScript implementations of the mathematical functions with the same names.

```
if (len(L) === 3n)
  return cons(1n, cons(2n, L));  // = swap(swap(cons(1, cons(2, L))))
```

The comment shows the definition of what should be returned, but the code is not a direct translation of those. Below, we will use reasoning prove that the code is correct.

(a) Let x be an integer. Prove that $\text{swap}(\text{swap}(x :: \text{nil})) = x :: \text{nil}$.

(b) Let x and y be integers and R be a list. Prove that

$$\text{swap}(\text{swap}(x :: y :: R)) = x :: y :: \text{swap}(\text{swap}(R))$$

(c) Let a, b, c, d, e be integers and $L = a :: b :: c :: d :: e :: \text{nil}$, i.e., L is some list of length 5. Prove that $\text{swap}(\text{swap}(L)) = L$.

You should apply part (a) once and part (b) multiple times (with different choices of x and y) rather than performing the same calculation again here. (Remember, that those facts we proved hold for *any* values of x and y.)

(d) Prove that the code is correct by showing that $\text{swap}(\text{swap}(1 :: 2 :: L)) = 1 :: 2 :: L$, using the fact that L has length 3, i.e., that $L = u :: v :: w :: \text{nil}$ for some integers u, v, w.

Feel free to apply prior parts, if useful, rather than performing calculations again.

Task 3 – Bright Glide and Bushy Tailed **[10 pts]**

In task 3 of Homework 4, we defined a function td : $\mathbb{Z} \to \mathbb{Z}$ that encoded characters (stored as integer values in the range 0–25) as other characters by shifting for characters in "toad" and negating for the others. Here, we provide a simpler cipher "glide" in which characters just glide to the other side of the alphabet (by 13 positions forward or backward depending on their starting point) instead of negating.

$$\text{glide} : \mathbb{Z} \to \mathbb{Z}$$

$$\begin{aligned}
\text{glide}(j) &:= j + 13 && \text{if } 0 \leqslant j \leqslant 12 \\
\text{glide}(j) &:= j - 13 && \text{if } 13 \leqslant j \leqslant 25 \\
\text{glide}(j) &:= j && \text{if } j < 0 \text{ or } 25 < j
\end{aligned}$$

For example 'i' (8) becomes 'v' (21), and 'w' (22) becomes 'j' (9). We claim that we can both encode characters and *decode* encoded characters back to their original value, using the same glide function[1]. In other words that, for an integer j

$$\text{glide}(\text{glide}(j)) = j$$

Prove this statement by cases.

Hint: If we know that $a \leqslant j \leqslant b$ and x is any integer, then we know that $a - x \leqslant j - x \leqslant b - x$ and $a + x \leqslant j + x \leqslant b + x$.

[1] Notice that we could do a similar proof by cases to prove that it's possible to use td to *decode* td-encoded characters also. However, because of the shifting applied to the characters of "toad", you must apply td <u>4 times</u> to return to the original values, rather than just once: $\text{td}(\text{td}(\text{td}(\text{td}(j)))) = j$. We will not make you prove this. (It's very long, you're welcome.)

Task 4 – Spice Apple Cipher [15 pts]

Similar to toad-cipher for td, we need an additional function to encode full messages (Lists of integers) where each character in the message is encoded with glide. We define glide-cipher as follows:

$$\text{glide-cipher} : \text{List}\langle\mathbb{Z}\rangle \to \text{List}\langle\mathbb{Z}\rangle$$

$$\begin{aligned} \text{glide-cipher}(\text{nil}) &:= \text{nil} \\ \text{glide-cipher}(j :: L) &:= \text{glide}(j) :: \text{glide-cipher}(L) \end{aligned}$$

We claim that we can both encode messages and decode encoded messages back to the original with glide-cipher[2]. You may cite that $\text{glide}(\text{glide}(j)) = j$ for some integer j as "Task 3", since we have already proven it.

Prove, by structural induction, that for any list S,

$$\text{glide-cipher}(\text{glide-cipher}(S)) = S$$

[2] Similarly, toad-cipher can be used to decode toad-cipher-encoded messages, but, like td, it requires applying toad-cipher 4 times: $\text{toad-cipher}(\text{toad-cipher}(\text{toad-cipher}(\text{toad-cipher}(S)))) = S$. We will not make you prove this either. (It's not that long, but still, you're welcome.)

The next problem will make use of some lists that do not contain integers. We can generalize our inductive List data type to allow it to store any type of data as follows:

$$\textbf{type } \mathsf{List}\langle T\rangle := \mathsf{nil} \mid \mathsf{cons}(\mathsf{hd}\colon T,\ \mathsf{tl}\colon \mathsf{List}\langle T\rangle)$$

A declaration like this is called a "generic" (or "parameterized") type. T is a *type* parameter, which we can fill in with any type we want. Filling in a different value for T gives us a different type. Hence, this one definition is creating infinitely many new types.

The type List$\langle T\rangle$ defines a list that stores elements of type T. The "hd" argument of cons is now a T rather than $\mathbb{Z}$. If we wish to have a list of integers, we would now write that as List$\langle\mathbb{Z}\rangle$.

The next problem will make use of the following functions that operate on the generic list type.

The function zip turns a pair of lists into a (single) list of pairs of numbers at the same positions in the two lists, is defined as follows:

$$\mathsf{zip} : (\mathsf{List}\langle\mathbb{Z}\rangle, \mathsf{List}\langle\mathbb{Z}\rangle) \to \mathsf{List}\langle(\mathbb{Z}\times\mathbb{Z})\rangle$$

$$\begin{aligned}
\mathsf{zip}(\mathsf{nil}, \mathsf{R}) &:= \mathsf{nil}\\
\mathsf{zip}(\mathsf{S}, \mathsf{nil}) &:= \mathsf{nil}\\
\mathsf{zip}(x :: \mathsf{S}, y :: \mathsf{R}) &:= (x, y) :: \mathsf{zip}(\mathsf{S}, \mathsf{R})
\end{aligned}$$

The function unpair turns a list of pairs into a list containing the numbers from the pairs in the order they were seen in the pairs, is defined as follows:

$$\mathsf{unpair} : \mathsf{List}\langle(\mathbb{Z}\times\mathbb{Z})\rangle \to \mathsf{List}\langle\mathbb{Z}\rangle$$

$$\begin{aligned}
\mathsf{unpair}(\mathsf{nil}) &:= \mathsf{nil}\\
\mathsf{unpair}((x, y) :: R) &:= x :: y :: \mathsf{unpair}(R)
\end{aligned}$$

The functions skip and keep make new lists containing every other element of the list, with skip skipping the first element (and keeping the second) and keep keeping the first element (and skipping the second). They are defined via mutual recursion as follows:

$$\mathsf{skip} : \mathsf{List}\langle\mathbb{Z}\rangle \to \mathsf{List}\langle\mathbb{Z}\rangle$$

$$\begin{aligned}
\mathsf{skip}(\mathsf{nil}) &:= \mathsf{nil}\\
\mathsf{skip}(x :: L) &:= \mathsf{keep}(L)
\end{aligned}$$

$$\mathsf{keep} : \mathsf{List}\langle\mathbb{Z}\rangle \to \mathsf{List}\langle\mathbb{Z}\rangle$$

$$\begin{aligned}
\mathsf{keep}(\mathsf{nil}) &:= \mathsf{nil}\\
\mathsf{keep}(x :: L) &:= x :: \mathsf{skip}(L)
\end{aligned}$$

Task 5 – Fish and Skips [15 pts]

So far, we've written a few super cool encoding and decoding functions for secret messages, but Matt says they are not secure enough ("everyone knows how to unshift!!"), so he has a scrambling cipher in mind that will really stump any malicious actors.

Here's the idea: put the characters at the even indices of the message in one list and the characters at the odd indices in another, then zip the lists together to create pairs, then turn it back into a message with unpair. In math, for the input message S, this would be $\mathsf{unpair}(\mathsf{zip}(\mathsf{keep}(S), \mathsf{skip}(S)))$.

James says, "But Matt, this won't scramble the message, it will just produce the original message S again!" Unfortunately Matt is not convinced. He says "I don't believe you, and I never will, unless 200 people write an induction proof verifying your claim!" Matt's the boss, so you'll have to help us out. Prove with structural induction that, for any $S \in \mathsf{List}\langle\mathbb{Z}\rangle$,

$$\mathsf{unpair}(\mathsf{zip}(\mathsf{keep}(S), \mathsf{skip}(S))) = S$$

One thing Matt doesn't need to be convinced of is that his encoding idea will only work well with messages that have an **even** number of characters, meaning that in your introduction you can state that your claim is "for any list $S \in \mathsf{List}\langle\mathbb{Z}\rangle$ such that $\mathsf{len}(S) \bmod 2 = 0$" rather than just "for any list"; the list in your inductive hypothesis should have the same condition. Then, in your inductive step, you should prove for a list with *2 more elements* than the list in your inductive hypothesis.

Coding

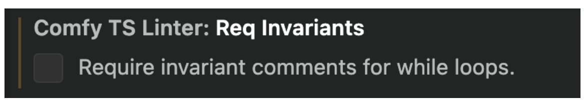
After finishing the written part, to get started on the coding part, check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25sp/materials/hw5-cipher.git
```

Navigate to the `hw5-cipher` directory and run `npm install --no-audit`. Run tests with the command `npm run test` and run the linter with the command `npm run lint`.

Starting with Homework 4, we **will not allow mutation** for a few assignments. The `npm run lint` command is already configured to disallow mutation. To disallow mutation with the VSCode extension, update the extension, restart VSCode, then open the extension, select the gear icon, open "Settings", and **uncheck** the checkbox on "Allow Mutation".

We'll also need to turn off invariant checking since we won't be covering it for a short while, so while you're at it also make sure to uncheck "Req Invariants".



Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW5 Code"**:

`cipher.ts` `cipher_test.ts` `App.tsx`

Wait after submitting to make sure the autograder passes, and leave yourself time to resubmit if there's an issue. The autograder will run your tests, additional staff tests, and the linter.

Task 6 – Let's Get This Show on The Code [10 pts]

In this problem, we will translate mathematical definitions for functions into TypeScript code.

We will treat the math definitions as the imperative specifications for the TypeScript functions, so the translations should be "straight from the spec" –a direct translation.

Unlike last week, we have not provided any tests for these functions. (You will write those in the next part!) However, you (or we) proved some claims related to these definitions in your written assignment, so if you translate those to code directly, you should already have some confidence that your code is correct.

(a) Implement td and toadCipher in TypeScript in `cipher.ts` by translating your mathematical definitions from HW4.

Complete the TODO in the `@returns` statement of the function specification above each of these functions by filling in the mathematical definition that you translated into TypeScript in the function body.

As a reminder, we originally described these functions in English and had you formalize them in HW4 Task 3. If you have since realized that you made a mistake in your original mathematical definitions, feel free to fix those when you type them up here. No additional explanation is needed, as we will refer to your comments to validate your translation.

(b) Implement glide and glideCipher in TypeScript in `cipher.ts` by translating the mathematical definitions from Tasks 3 and 4.

Note the `@returns` statement in the function specification also contains its mathematical definition that you should translate into TypeScript in the function body.

(c) For our last cipher, we are ignoring our failed attempt from Task 5, and wrote a brand-new scrambling cipher. crazyCipher takes a list of characters, swaps the second and third characters and repeats on the remainder of the list following the 4th character.

Like the others, crazyCipher-encoded messages can be decoded by applying crazyCipher again[3]. To really understand what this function is doing, we recommend playing around with some examples.

We have provided a definition of crazyCipher as a comment spec above the function in `cipher.ts`. Implement crazyCipher in TypeScript in `cipher.ts` by translating the mathematical definition in the `@returns` statement in the function specification.

Task 7 – No Test For The Wicked [10 pts]

Now for our final step in gaining confidence our code is correct, it's time to test the functions we translated in the last part!

Write tests for `td`, `toadCipher`, `glide`, `glideCipher` and `crazyCipher` in `cipher_test.ts`.

Your tests should follow the testing requirements for this course (see the notes on testing for a reminder). Additionally, write short labels describing which coverage requirement is met by each test.

[3] We won't make you prove it, we already did!

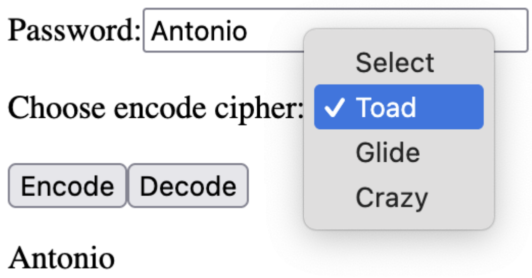
See the example from HW4 for reference.

We also wrote tests for these functions which the autograder will run on your code. Some test results will be *hidden* until after the deadline (meaning, passing the autograder before the deadline is not a guarantee of full points on staff tests). **So we urge you to test your functions well!**

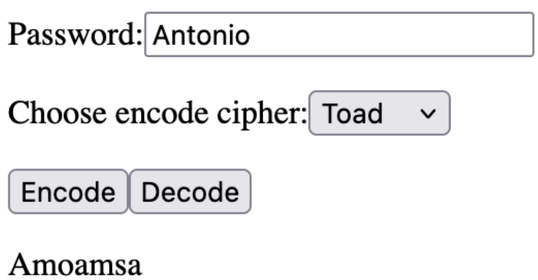
Task 8 – Fell Right into My App [10 pts]

We decided to use these cipher functions in a password encoding app for some extra security. You will write this app in `App.tsx`. As a reminder, you can run the app with the command `npm run start`.

First, we need a page for entering a password and a way to select how we want to encode it. As inspiration for how this can look, see the screenshot of our app below.



When the user clicks "Encode", the app should replace the password with the encoded version of the password, and show a "Decode" button to view the original version again. You should use the functions you implemented in Task 6 to encode and decode according to the users selected cipher. As inspiration for how this changed view can look, see the screenshot of our app below.



Along with these design ideas for the app, we've also provided some list functions to streamline processing your passwords between string messages and lists of integers (which the cipher functions take as input and produce as output) in `list_ops.ts`.

Remember to test your app or have someone else try it out before turning in!

Congrats! You have created another amazing app!